

A Lightweight and Portable Approach to Making Concurrent Failures Reproducible

Qingzhou Luo, Sai Zhang, Jianjun Zhao
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
{seriousam, saizhang, zhao-jj}@sjtu.edu.cn

Abstract

Multithreaded concurrent programs often exhibit bugs due to unintended interferences among the concurrent threads. Such bugs are often hard to reproduce because they typically happen under very specific interleaving of the executing threads. Basically, it is very hard to fix a bug (or software failure) in concurrent programs without being able to reproduce it. In this paper, we present an approach, called ConCrash, that automatically and deterministically reproduces concurrent failures by recording logical thread schedule and generating unit tests. For a given bug (failure), ConCrash records the logical thread scheduling order and preserves object states in memory at runtime. Then, ConCrash reproduces the failure offline by simply using the saved information without the need for JVM-level or OS-level support. To reduce the runtime performance overhead, ConCrash employs a static datarace detection technique to report all possible race conditions, and only instruments such places. We implement the ConCrash approach in a prototype tool for Java and experimented on a number of multithreaded Java benchmarks. We successfully reproduced a number of real concurrent bugs (e.g., deadlocks, data races, two-stage access) within an acceptable overhead.

1 Introduction

The increasing popularity of multithreaded concurrent programming has brought the issue of concurrent defect analysis to the forefront. Multithreaded concurrent programs often exhibit wrong behaviors due to unintended interferences among the concurrent threads. Such concurrent failures or bugs - such as data races and deadlocks - are often difficult to fix without being able to reproduce them. However, in a multithreaded concurrent program, the number of possible interleavings is huge, and it is not practical

to try them all. Only a few of the interleavings or even one specific interleaving actually produce the failure; thus, the probability of reproducing a concurrent failure is extremely low. A traditional method of reproducing concurrent failure is to repeatedly execute the program with the hope that different test executions will result in different interleavings. There are several problems with this approach. First, program executions revealing a concurrent failure are usually long and run under different environmental conditions. Second, execution result depends on the underlying operating system or the virtual machine for thread scheduling - it does not try to explicitly control the thread schedules; therefore, executions often end up with the same interleaving many times. As a result, executing concurrent programs repeatedly is not necessary, and when a failure is detected, much effort must be invested in reproducing the conditions (e.g., thread interleaving orders) under which it happens.

The high cost of reproducing concurrent failures has motivated the development of sophisticated and automated analysis techniques, such as [8–10, 12, 13, 17, 20, 22, 25]. Of particular interest for our work is the ReCrash approach proposed by Artzi et al [7]. In the pioneering work [7] on reproducing a software failure. They monitor every execution of the target(sequential) program, store partial copies of method arguments, and convert a failing program execution into a set of deterministic unit tests, each of which reproduces the problem that causes the program to fail. The ReCrash approach is designed for sequential programs. However, the non-determinism in a multithreaded concurrent program might disallow the unit tests generated by ReCrash to reproduce a concurrent failure.

The work described in this paper aims to reduce the amount of time a developer spends on reproducing a concurrent failure. A key element in designing such an approach is the ability to provide a deterministic thread executing order of a non-deterministic execution instance. In this paper, we propose ConCrash, an automated concurrent failure reproducing technique for Java. ConCrash handles all threads

and concurrent constructs in Java, except for windowing events, I/O inputs and network events which are topics of our future work.

The ConCrash approach adapts the concept of *logical thread schedule* as described in [9]. It monitors each critical event to capture the thread execution order during one execution of a multithreaded program. When a concurrent program fails, ConCrash saves both information about thread scheduling and current object states in memory, and automatically generates an *instrumentation scheme* and a set of *JUnit tests*. The *instrumentation scheme* records the thread schedule information during the failing execution as pure text, and then enforces the exact same schedule when replaying the execution. The ConCrash approach can be used on both client and developer sides. When a concurrent failure occurs, the user could send a *instrumentation scheme* as well as generated *JUnit tests* to developers. While developers could use a ConCrash-enabled environment to replay the thread execution order, step through execution, or otherwise investigate the cause of failure.

Unlike most of the existing replay techniques [9], our ConCrash approach does not depend on JVM modification or existing OS-level support for replay. Instead, ConCrash instruments the compiled class files by modifying its bytecode. To reduce the runtime performance overhead, ConCrash also employs a static datarace detection technique [17] to find all possible race conditions, and only instruments such places. While starting to reproducing a failure, ConCrash eliminates the program’s nondeterminacy caused by JVM scheduler by transforming the compiled nondeterministic multithreaded program into a deterministic sequential program without changing the semantics.

We implement the ConCrash approach in a prototype tool for Java and experimented on a number of multithreaded Java benchmarks. We successfully reproduced a number of real concurrent bugs (e.g., deadlocks, data races, atomicity violation) within an acceptable overhead. The main contributions of this paper are:

- A lightweight and portable technique that efficiently captures and reproduces multithreaded concurrent failures.
- Implementation of a prototype tool for Java.
- An empirical evaluation that shows the effectiveness and efficiency of ConCrash approach on Java benchmarks and real-world applications.

The rest of this paper is organized as follows. In Section 2, we give an overview of the ConCrash approach using a simple motivating example. We describe the details of the ConCrash approach in Section 3. In Section 4 and Section 5, we describe the implementation issues of ConCrash approach for Java and the results of our experiments, respectively. Related work is discussed in Section 6 followed

by conclusion, where we recommend our ConCrash approach as a supplementary part of the existing ReCrash approach [7] when software failure occurs.

2 Motivating Example

In this section, we use a real-world program to give an overview of our approach. Consider the two-threaded program snippet taken from *hedc* benchmark [17] in Figure 1.

Two threads executing the code of *MetaSearchResearch.java* and *Task.java* have one shared variable *thread_*. There could be an unsynchronized assignment of null to field *thread_* (line 55 of *Task.java*), which could cause the program to crash with a *NullPointerException* (line 54 in *MetaSearchResult.java*) if the *Task* completes just as another thread calls *Task.cancel()*.

```

In MetaSearchResult.java
51.   public synchronized void cancel() {
52.       if(thread_ != null) {
53.           thread_.interrupt(); //Potential crash point
54.       }
55.   }

In Task.java
49.   public void run() {
50.       try { runImpl(); }
51.       catch (Exception e) {
52.           Messages.warn(-1, "Task::run exception=%1", e);
53.           e.printStackTrace();
54.       }
55.       thread_ = null;
56.   }

```

Figure 1. A Motivating Example

In this case, it would be nearly impossible to reproduce the failure by repeatedly executing the original program due to the huge volume of different thread execution orders. Moreover, since in typical OS and JVM design, the thread scheduler is nearly deterministic, executing the same program many times does not help, because the same interleaving is usually created. Actually, in our experiment environment (Section 5), executing the original program for more than 300 times could reproduce only one failure.

Suppose that the motivating example is run in a ConCrash-enabled environment. ConCrash first uses an existing static datarace detection technique [17] to preprocess this program to find all possible race conditions (it is a one time cost). ConCrash instruments the program at the reported race conditions. In this example, line 53 in *MetaSearchResult.java* and line 55 in *Task.java* are reported to be potential race conditions and ConCrash instruments these two statements to monitor the thread execution logical order. For each method invoked, ConCrash also maintains a state copy of the method receiver and arguments.

As soon as the (instrumented) program crashes, ConCrash will generate an *instrumentation scheme* and a set of *JUnit tests*. The user could send the scheme and tests with the initial bug report to developers. Upon receiving such a scheme and tests, the developer could use ConCrash (we provide an instrumentation tool in ConCrash design) to instrument the original program to resume the original thread schedule order. The purpose of this instrumentation step is to convert the non-deterministic multithreaded program into deterministic sequential program without changing its semantics. After then, the developer could run tests under a debugger to easily reproduce the failure and locate its cause.

For this motivating example, one of the generated JUnit tests is shown in Figure 2. In line 2 - 6 of Figure 2, ConCrash first reads the current object (`thisObject`) from a trace file to resume the state. This part is adapted from the existing `reCrashJ` implementation [4]. ConCrash then reads other thread objects from recorded file (line 7, 8), and synchronizes them to restart at a certain checkpoint before crash occurs (line 9, 10). Finally, ConCrash invokes the crashed method (and load arguments if there is any) to reproduce the failure.

```

1. public void MetaSearchResult_cancel_3() throws Throwable {
2.     //Read object from trace file
3.     //adpated from ReCrash implementation
4.     TraceReader.setMethodTraceItem(3);
5.     MetaSearchResult thisObject =
6.         (MetaSearchResult) TraceReader.readObject(0);

7.     // Resume thread execution orders
8.     ThreadEntity te =
9.         TraceReader.getStackTraceItem().threadEntity;
10.    Monitor.restartThreads(te.checkPoints,3);
11.    Monitor.waitForThreads();

12.    // Method invocation
13.    thisObject.cancel();
14. }

```

Figure 2. A generated test case by ConCrash to reproduce the hedc failure.

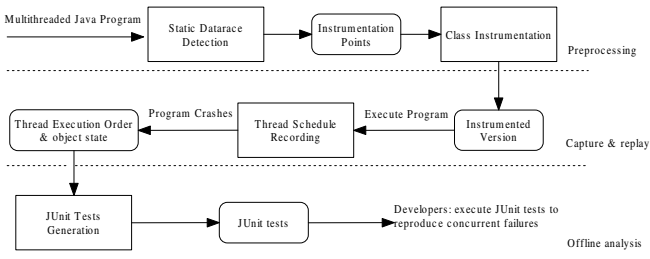


Figure 3. An overview of the ConCrash approach

3 Approach

The overview of our approach is shown in Figure 3. Our approach consists of three stages: preprocessing (Section 3.1), record & replay (Section 3.2), and offline analysis (Section 3.3). The preprocessing stage employs an existing static datarace detection technique [17] to find all possible race conditions. Then, only a small part of program will be instrumented. The record & replay stage monitors the thread logical execution orders and preserves a state copy of each object at runtime. The offline analysis stage synergizes the information collected during program execution and generate corresponding failure-reproducing JUnit tests.

3.1 Preprocessing

The purpose of preprocessing stage is to reduce the performance overhead. As described in literature [15, 19], most of the concurrent failures are caused by data races. In the ConCrash approach, we employ Chord to statically report all possible race conditions, and only instrument on such program parts.

3.1.1 Static Datarace Detection

Monitoring all program statements to capture the thread execution orders will introduce a significant performance overhead. To eliminate such cost, we employ an existing datarace detection tool (Chord [17]) in our approach to detect potential dataraces. A multithreaded program contains a race if two threads can access the same memory location without ordering constraints enforced between them and at least one access is a write. As described in literature [15, 19], race conditions are often considered to be manifestations of bugs and different access orders for a race variable may cause the program behave differently. We use Chord to report potential race conditions, and focus on such conditions when perform instrumentation.

Chord is a static race detection tool for Java programs. The key point of Chord is the *k-object sensitivity* alias analysis, and its detection algorithm is context sensitive and flow insensitive. Chord reads program's source code and byte code, performs four stage analysis and outputs the results in files. Though it reports some false positives, its experimentation shows that Chord is applicable for static race detection for most programs.

► **Example.** For the motivating example in Figure 1, Chord reports one data race pair, line 53 in `MetaSearchResult.java` and line 55 in `Task.java` on the `thread.shared` variable. ◀

3.1.2 Instrumentation Points

Instead of instrumenting the whole program, we use the potential races output by Chord and only instrument on these points.

We insert **monitorenter** instruction before each instrumentation point(IP), to guarantee the atomic execution for the statement in that point. The instruction is associated with a special lock object. After acquiring the lock, the current thread executes the critical event in the IP. After successful execution, it increases global clock and its own local clock. At the same time, the thread checks whether its local clock equals with the global clock. If not, it indicates that at least one thread switch happened after last critical event's execution in current thread. In that situation we synchronize the local clock with the global clock, record the last local clock value and generate a interval of thread execution trace.

►**Example.** For the motivating example in Figure 1, ConCrash only instruments before and after the statements which contain potential datarace (line 53 in `MetaSearchResult.java` and line 55 in `Task.java`) The instrumented code sample is shown in Figure 4. Note that, adapting the technique used in ReCrash [7], we also instrument the entry and exit point of each method to store the receiver and the arguments on the shadow stack. ◀

```
In MetaSearchResult.java
51.   public synchronized void cancel() {
      int _id = ShadowStack.push("cancel");
      ShadowStack.addReceiver(this);
52.   if(thread_ != null) {
      synchronized(Monitor.lock)
      {
53.       Monitor.recordGlobalClock();
      thread_.interrupt(); //Potential crash point
      Monitor.recordGlobalClock();
      }
      ...
    }
    ShadowStack.popUntil(_id);
  }

In Task.java
50.   public void run() {
      int _id = ShadowStack.push("run");
      ShadowStack.addReceiver(this);
51.   try { runImpl(); }
52.   catch (Exception e) {
53.       Messages.warn(-1, "Task::run exception=%1", e);
53.       e.printStackTrace();
54.   }
      synchronized(Monitor.lock)
      {
55.       Monitor.recordGlobalClock();
      thread_ = null;
      Monitor.recordGlobalClock();
      }
      ShadowStack.popUntil(_id);
56.   }
```

Figure 4. The instrumented code of Figure 1 to record logical thread order. Instrumentation code is bold.

3.2 Record & Replay

As described before, to capture a thread's schedule, we do not need to record every event during its execution. Instead, we only care about the time intervals which are bounded with thread switch points.

3.2.1 Logical Thread Schedule

Capturing the actual physical thread schedule information is neither feasible nor useful in our approach. Rather than doing so, we record the thread's 'Lamport clock' as a logical thread schedule. The Lamport clock was proposed by Lamport [14], which used a scalar time stamp to measure the logical clock of each thread execution. For example, we use global clock and local clock to denote the time stamp of the global program and each thread respectively. At the beginning the global clock and all the threads' local clock are initialized with 0. Whenever a thread performs a synchronization operation, its Lamport clock value is updated due to the thread's local clock and the global clock

$$LC_{new} = \max(LC_{local}, LC_{global}) + 1 \quad (1)$$

Then the new clock value is assigned to current thread's local clock value and the global clock value at the same time.

During the record phase, we record the time stamp of every *critical event* in each thread's execution, which in Java are consisted of:

Shared variable accesses are essential for dividing thread schedules into different equivalent classes. We use the term 'shared variable' to represent the variable that can be accessed by at least two different threads, which contrasts with thread-local variable that could only be accessed in one thread. When multiple threads access the thread-shared variables in different order, the program may behave differently.

Synchronization events could potentially affect the order of shared variable accesses, thus affecting the possible logical thread schedule. Java provides several kinds of synchronization operations: **monitorenter** and **monitorexit** are used to acquire and release a lock of an object respectively, which mark the begin and end of a critical section in the program. To enter a critical section, a thread needs to acquire the specified lock using **monitorenter**, and release the lock after execution with **monitorexit**; **wait** and **notify/notifyAll** can be used to adjust the thread execution order; **suspend** and **resume** are also used for the same purpose, and finally **interrupt** can be used to interrupt another thread's execution. All the above synchronization events could affect the thread execution order and need to be reproduced as the interactions between threads.

Note that we do not need to trace every individual critical event in thread execution, instead we use a logical thread schedule to record the begin and the end time stamps

of a few of consecutive critical events in one thread. In each thread schedule, we record information in the form $\langle \text{FirstCriticalEvent}, \text{LastCriticalEvent} \rangle$. The interval in every tuple is the thread's logical running time before a thread switch occurs.

3.2.2 Thread Execution Order Record & Replay

In each method entry, we make a checkpoint for all current running threads. Whenever a new thread enters a method, it is registered on a global thread map. Each checkpoint maintains the information of a thread's executing method items and the number of instrumentation points it has executed inside the current method. We use a checkpoint stack for each thread to push and pop checkpoint when entering or existing a method. When a crash happens, the checkpoint stack will be serialized and written to disk.

When replaying a crash, the checkpoint information will be used to recreate thread state. When implementing this, a wrapper thread is introduced (discussed later). We count the number of instrumentation points each replaying thread executed to determine whether they reach the same state before the crash. After all replaying threads reach their own recorded instrumentation point, the crashed thread will be invoked. Figure 5 shows the instrumented program for replay.

```
In MetaSearchResult.java
51.  public synchronized void cancel() {
52.      if(thread_ != null) {
53.          synchronized(Monitor.lock)
54.          {
55.              while(Monitor.globalClock < getFirstCriticalEvent(num))
56.                  Monitor.lock.wait();
57.              thread_.interrupt(); //Potential crash point
58.              if(Monitor.globalClock >= getLastCriticalEvent(num))
59.                  num++;
60.              Monitor.globalClock++;
61.              Monitor.lock.notifyAll();
62.          }
63.      }
64.  }
65.  ...
66. }
```

```
In Task.java
49.  public void run() {
50.      try { runImpl(); }
51.      catch (Exception e) {
52.          Messages.warn(-1, "Task::run exception=%1", e);
53.          e.printStackTrace();
54.      }
55.      synchronized(Monitor.lock)
56.      {
57.          while(Monitor.globalClock < getFirstCriticalEvent(num))
58.              Monitor.lock.wait();
59.          thread_ = null;
60.          if(Monitor.globalClock >= getLastCriticalEvent(num))
61.              num++;
62.          Monitor.globalClock++;
63.          Monitor.lock.notifyAll();
64.      }
65.  }
```

Figure 5. The instrumented program of Figure 1 for replay. Instrumentation code is bold.

3.3 Offline Analysis

After a multithreaded program crashes, we use the information recorded to generate test cases and recreate all the threads to simulate the crash scenario. Besides, we also provide threads' execution traces based on instrumentation points to describe thread schedules for developers.

3.3.1 Unit Tests Generation

We augment the unit tests generation technique [7] developed by Artzi et al. to handle concurrent features. In their approach, the method call stack data is captured and used for test cases generation when a crash happens. Whenever a method is called, they push the receiver object and parameters onto stack, and then pop those information after normal exit of the method. After a crash happens, ReCrash can simply generate test cases by passing the same receiver object and method parameters, which are serialized to the disk before. However, the ReCrash can not be applied to concurrent programs for the lack of monitoring threads' states. ReCrash only monitors main thread in the program execution, while exception thrown by other threads couldn't be monitored and handled by main thread.

As reCrashJ only supports crash exceptions in main thread, we extend it to support all the running threads. Each thread begins with the run() method, so ConCrash replaces the original run()'s name with a new name and adds a new run() (shown in Figure 6).

```
class Task
{
    ...
    public void run()
    {
        try
        {
            __original_run__benchmarks_hedc_Task();
        }
        catch(Throwable e)
        {
            e.printStackTrace();
            System.out.println(e.getMessage());
            TraceWriter.writeTrace(e);
        }
    }
    ...
}
```

Figure 6. Instrumentation code for original run method.

All the threads run concurrently in multithreaded programs. For the purpose of reproducing concurrent program's crashes, only the crashed thread's method call stack data is not sufficient. Besides, we also need all other threads' execution information to recreate these threads and trigger the crash exception cooperatively. To do this, we make checkpoints at the time when any thread enters any

method. The checkpoint contains all the alive threads' execution information, which consist of the method's name, the thread that is executing, and the number of critical events the thread has executed in that method. The checkpoints are stored together with the thread's method stack data, so they can be deserialized in test case generation stage. We will show that with the checkpoints' information we are able to simulate the scenario of multithreaded crashes.

When generating test cases, we not only load objects to invoke the crashed method, but also invoke all other alive threads at the crash time. To make sure all threads reach the same points as in the crash scenario, we count the instrumentation points it has executed, compare them to the recorded information. If a thread has executed the same amount of instrumentation points recorded before, then we simply consider it has reached the same state as in the crash point.

►**Example.** For the motivating example in Figure 1, suppose when thread T1 in `MetaSearchResult.java` enters the method `cancel`, another thread T2 in `Task.java` has just finished the invocation of `runImpl()`. In the method entry of `cancel` we record the extra information about the name of method T2 is in (In this example is `Task.run`), and also the number of critical events T2 has executed in `runImpl()`. With these information, we could recreate threads and let them reach the same state in crash time. ◀

The last issue we care about is how to recreate all corresponding threads in the generated test cases. Since JUnit doesn't support multithreaded test cases up now, we use a *WrapperThread* to achieve the goal. By using Java Reflection APIs, the *WrapperThread* has the ability to dynamically invoke the specified method when needed. Figure 7 shows the code of reflection in *WrapperThread*.

```
public void run()
{
    ...
    try
    {
        Method m = c.getDeclaredMethod(methodName, paraTypes);
        m.invoke(this.owner, this.args);
    }
    catch (NoSuchMethodException e)
    {
        System.out.println("Error");
        return;
    }
    ...
}
```

Figure 7. Code sample for *WrapperThread*.

After all threads are recreated, we let them run arbitrary until they execute the same number of critical events recorded in the checkpoint before. We set a *barrier* for all the threads, when they all reach the same state as in the checkpoint, we release the *barrier* and deterministically replay their execution based on the recorded schedule. Then the crash will be triggered again with all threads' cooperation.

4 Implementation

We have implemented our approach for concurrent Java programs using the Chord static datarace detection tool [17], reCrashJ tool [7], XStream [5] framework, and ASM library [1]. The current implementation of our approach supports Java version 1.6. We next present the implementation details of the main stages of our approach.

Preprocessing stage. We have implemented the preprocessing stage based on the Chord [2], an existing static datarace detection tool for Java. Chord takes the program's source code and bytecode as an input, uses call-graph construction, alias analysis, thread-escape analysis and lock analysis to compute pairs of memory accesses involved in races. We store the Chord's running results in a file as a list of instrumentation points, each of which specified by the class name involved in race, the line number of the race statement and the field of race variable.

Record & replay stage. In this stage, we implement the instrumentation engine in the ConCrash approach using ASM [1]. Instead of modifying the underlying JVM or performing source-code-level instrumentation, we only instrument pure Java bytecode. We use synchronized code block to wrap each statement in instrumentation points, and then synchronize local logical clock with global logical clock and finally increase global clock after successfully execute the statement. Figure 8 shows the instrumented code for the statement "times++".

```
synchronized(Monitor.lock)
{
    gc_copy = Monitor.globalClock;
    Monitor.globalClock++;
    if (localClock < gc_copy)
    {
        LastCriticalEvent[num] = localClock;
        num = num + 1;
        FirstCriticalEvent[num] = gc_copy;
    }
    ++ times;
    localClock = gc_copy + 1;
}
```

Figure 8. Instrumentation code for synchronization.

When recording the state objects, ConCrash keeps a *shadow copy* of each object on the shadow stack.

Offline analysis. We implement the test generation part by modifying reCrashJ [4] to support multithreaded features. Like reCrashJ, the ConCrash implementation uses the stored shadow stack to generate a suite of JUnit tests. Each test in the suite invokes all alive threads at the checkpoints, and loads the receiver and method arguments from the serialized shadow stack.

5 Empirical Evaluation

To evaluate the effectiveness and efficiency of our proposed technique, we experimented on a number of multi-threaded Java benchmarks. Our experimental results indicate that ConCrash is able to deterministically reproduce a number of typical concurrent failures with a tolerable overhead. For each benchmark, we also analyze the reproduced bugs in details and show how the generated test cases are useful for debugging.

5.1 Subject Programs

Programs	#Loc	#Class	#Method	#Threads
Allocation Vector	304	3	7	3
Xtango	2088	31	220	3
Hedc	29948	530	3552	8

Table 1. Subject Programs

We evaluate ConCrash on a suite of multi-threaded Java programs. The suite includes the multi-threaded benchmarks from the Java Grande suite (AllocationVector [3]), an animation library (Xtango [23]), and a web crawler (hedc [17,25]).

Table 1 summarizes the number of lines of code in the original program (#Loc), the number of classes (#Class), the number of methods (#Method), and the number of active threads (#Threads).

5.2 Procedure

We focus on the crash reproducibility and performance overhead of ConCrash in this empirical study. For each subject program, we first use Chord to find all potential datarace conditions. We instrument the program and deploy it into a ConCrash-enable environment.

We follow the instructions on bug report [2] to repeatedly run the program, to make the concurrent failure happen. When a buggy interleaving happens and the program crashes with an exception, ConCrash output an instrumentation scheme which captures the thread execution orders (when failure occurs), and a set of JUnit tests.

To evaluate the reproducibility of generated test cases, we first use ConCrash to instrument the subject program using the generated instrumentation schedule to make the multithreaded program to run deterministically. Then, we run all generated tests (on the instrumented version) to check whether the exact same exception will be thrown. We also measure and compare the time and memory usage of the original and instrumented versions of subject program, while reproducing a concurrent failure.

5.3 Results

The experiment results of concurrent crash reproducibility and performance overhead are shown in Table 2 and Table 3, respectively. In Table 2, column #Tests is the total number of JUnit tests ConCrash has generated, while column #Rep is the number of JUnit tests which could reproduce the crash. In Table 3, column *Original(ms)* is the execution time of original program, column *Instrument(ms)* is the execution time of ConCrash-instrumented version of subject program, and column *File size (kb)* is the size of generated trace file (including instrumentation scheme).

Programs	Bug Type	Exception Type	#Tests	#Rep
Allocation Vector	Two-stage access	inconsistent	2	2
Xtango	Deadlock	-	1	1
Hedc	Datarace	null pointer	4	3

Table 2. Crash reproducibility study result.

5.3.1 Failure Reproducibility

We can observe from Table 2 that ConCrash was able to reproduce all three kinds of typical concurrent crashes (Two-stage access, deadlock, and datarace) in the subject program investigated. For the crashes in Allocation Vector and Xtango benchmarks, ConCrash achieves a 100% reproducibility rate. For the Hedc benchmark, 3 out of 4 generated test cases can reproduce the crash caused by data race.

Allocation Vector benchmark. This program performs management of allocated and deallocated blocks, by using Allocation Vector/BitVector/itmap, in which i-th entry indicates the status of i-th block. There is a two-stage access bug in line 48 - 50 in TestThread1.java file. When block is allocated, first "free block index found" and then "the free block is marked as allocated", and this two actions are not synchronized between them, therefore a user-defined state inconsistency will be thrown. ConCrash generates two tests to reproduce this failure. We show one of them as follows in Figure 9.

Xtango benchmark. Xtango comes with a deadlock bug. It is a demo program that among other things create two circle which calls exchangePosAsync and exchangePos. Both methods cause the two circles to exchange position by sidling across the screen. However, this can cause deadlock, because exchangePos and exchangePosAsync both lock the first circle object then the second. Without the help of ConCrash, repeatedly execution 200 times could lead to only 1 deadlock. In contrast, ConCrash achieve a 100% reproducibility rate.

Hedc benchmark. The hedc benchmark is developed by

```

public void(testMarkAsAllocatedBlock)
{
    TraceRead.setMethodTraceItem(1);
    Bug.AllocationVector thisObject
        = (Bug.AllocationVector) TraceReader.readObject(0);

    //Replay for multiple threads
    ThreadEntity te =
        TraceReader.getStackTraceItem().threadEntity;
    Monitor.restartThreads(te.checkPoints, 1);
    Monitor.waitForThreads();

    //load arguments and invoke method
    int arg_1 = 6;
    thisObject.markAsAllocateBlock(arg_1);
}

```

Figure 9. Test case generated for Allocation Vector benchmark.

ETH [25] to implement a meta-crawler for searching multiple scientific Internet archives in parallel. This benchmark includes three threads in total, two of which interact with each other to issue and handle random queries. It includes a bug as we described in Figure 1. While this bug occurs under one very specific thread interleaving, it would be nearly impossible to reproduce it merely by repeated executions. ConCrash generates four tests (Figure 2), one of which fails to reproduce the failure. We will discuss the reason later.

5.3.2 Performance Overhead

Programs	Original(ms)	Instrument(ms)	File size(kb)
Allocation Vector	32	1313	12
Xtango	-	-	7
Hedc	7904	8804	27

Table 3. Performance overhead study result.

We compare the execution time of original program and ConCrash-instrumented version in Table 3. The instrumented version of Allocation Vector imposes a slow down about 40 times. That is because the shared variable involved in the two-stage access bug is in a deep loop, so we need to checkpoint every time when the loop is executed. Although this overhead is significant, it has not limited our ability to collect data from long-running program. The Xtango benchmark suffers a deadlock bug, so we could not measure the accurate execution time. There is a relative execution time slowdown (17%) in the Hedc benchmark. As we see in the motivating example, the datarace bug in Hedc occurs in `run()` and `cancel()` methods. Thus, we only need to instrument before and after these two places.

The trace file size including instrumentation scheme is very small - 27kb for the largest benchmark. It could be easily sent via Internet to developers when crash happens.

5.4 Threat to Validity

Like any empirical evaluation, this study also has limitations which must be considered. Although we have experimented several well-known multithreaded Java benchmarks, in which the largest one is over 29KLOC, they are smaller than traditional Java software systems. For this case, we can not claim that these experiment results can be necessarily generalized to other programs. On the other hand, the systems and crashes we chose to investigate might not be representative. Though we experimented on reproducing several typical concurrent failures, such as data race, deadlock, and atomicity violation, we still can not claim ConCrash could reproduce an arbitrary crash in concurrent software.

Another threat is that we have not conducted any readability study about the generated test case on software developers (thought the readability of the test case is secondary to reproducibility). For this reason, we might not be able to claim ConCrash can be applied to real-world development process.

The final threat to internal validity maybe mostly lie with possible errors in our tool implementation and the measure of experiment result. To reduce these kind of threats, we have performed several careful checks.

5.5 Discussion

When designing the ConCrash approach, we choose bytecode level instrumentation to make concurrent failure be deterministically reproduced. If we use OS-level support or JVM-level capture and replay, ConCrash would have to be deployed on a specific environment, which will deteriorate the portability of our approach. In current ConCrash implementation, no other program or configuration is needed, and the pre-instrumentation also permits the high comparability of ConCrash-enabled environment.

As mentioned above, we use Chord to report potential datarace conditions in order to reduce the performance overhead. However, the result of Chord (or other analysis techniques used for preprocessing) might affect the precision/effectiveness of the ConCrash approach. Investigating the tradeoffs would be one of our future directions.

Current implementation of ConCrash reuses the reCrashJ infrastructure. It uses shallow (depth-1) copying strategy as default mode. However, in some cases (e.g., the Hedc benchmark) an argument is side-effected, between the method entry and the crash point, in such a way that will prevent the crash from reproducing.

6 Related Work

In this section, we discuss some closely related work in the areas of multithreaded program analysis, testing, and

debugging.

Much research has been done on testing and debugging multithreaded programs. Researchers have proposed analysis techniques to detect deadlocks [18], data races [17], and atomicity violations [12]. The problem of generating different interleavings for the purpose of revealing concurrent failures [18] and record/replay techniques [9, 16, 24] have also been examined. Moreover, systematic and exhaustive techniques, like model checking [13], have been developed recently. These techniques exhaustively explore all interleavings of a concurrent program by systematically switching threads at synchronization points.

Choi et al. [9], presented the concept of logical thread schedule and proposed an approach to deterministically replay a multithreaded Java program. They are able to reproduce race conditions and other non-deterministic failures. However, Choi et al.'s method relies on the modification of underlying JVM, while our method uses bytecode level instrumentation to capture the thread execution orders. Moreover, our approach generates a series of JUnit tests, which helps developers to confirm the concurrent failures.

The most similar work to us is Kim et al. [7]'s pioneering ReCrash approach. ReCrash generates tests by utilizing partial snapshots of the program states captured on each method execution in the case of a failure. The empirical study shows that ReCrash is simple to implement, scalable to large programs, and generates simple but helpful tests. Our work on ConCrash aims to extend the ReCrash technique to handle concurrent failures. We use the concept of logical thread schedule to capture the execution order, making the behavior of multithreaded programs deterministic when reproducing the concurrent failure.

There is also a rich body of work devoted to the problem of automated test generation. jRapture [24], test factoring [21], and test carving [11] capture the interactions between the program and the environment to create smaller test cases to reproduce the same failures. These techniques utilize a trace, and then run the subject program in a special harness, such as a mock object representing all interactions with the rest of the system. In contrast, ConCrash monitors the thread execution order and generates tests that reproduce real crashes. Like ReCrash, ConCrash could also be used in either in-house or in-field. The trace file recorded by ConCrash is much smaller than a core-dump or even crash reporting systems and thus is easier to send to the developers.

7 Concluding Remarks

In this paper, we presented a lightweight and portable approach, called ConCrash, to make concurrent failures reproducible. ConCrash records the logical thread scheduling order and preserves object states in memory at runtime.

When a crash occurs, it reproduces the failure offline by simply using the saved information without the need for JVM-level or OS-level support. To reduce the runtime overhead, ConCrash uses an effective existing data race detection technique to report all possible race conditions, and only instruments such places. We implement the ConCrash approach in a prototype tool for Java. Our experiment on several well-known multithreaded Java benchmarks indicates that ConCrash is effective in reproducing a number of typical concurrent bugs within an acceptable overhead.

We recommend the ConCrash approach be an integrated part of the existing ReCrash technique. As our future work, we would like to examine alternative techniques like dynamic program slicing [6] to improve the performance of ConCrash. We also intend to investigate the cost/effectiveness tradeoffs when reproducing concurrent failures at the application level.

Acknowledgements. This work was supported in part by National High Technology Development Program of China (Grant No. 2006AA01Z158), National Natural Science Foundation of China (NSFC) (Grant No. 60673120), and Shanghai Pujiang Program (Grant No. 07pj14058). We would like to thank Cheng Zhang and Feng Xie for their valuable discussions on this work.

References

- [1] ASM Homepage. <http://asm.objectweb.org/index.html/>.
- [2] Chord Project Homepage. <http://code.google.com/p/jchord/>.
- [3] IBM Haifa Research Lab. <http://www.haifa.ibm.com/research.html>.
- [4] reCrashJ implementation. <http://groups.csail.mit.edu/page/reCrash/>.
- [5] XStream Project Homepage. <http://xstream.codehaus.org/>.
- [6] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM.
- [7] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP '08*, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.
- [8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise data race detection for multithreaded object-oriented programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, New York, NY, USA, 2002. ACM.
- [9] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, New York, NY, USA, 1998. ACM.
- [10] J.-D. Choi and A. Zeller. Isolating failure-inducing thread schedules. volume 27, pages 210–220, New York, NY, USA, 2002. ACM.
- [11] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–264, New York, NY, USA, 2006. ACM.

- [12] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.
- [13] S. N. Freund. Checking concise specifications for multithreaded. In *Journal of Object Technology*, volume 3, page 81101, 2004.
- [14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [15] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Comput. Archit. News*, 36(1):329–339, 2008.
- [16] M. Musuvathi, S. Qadeer, G. B. T. Ball, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. *OSDI'2008*, 2008.
- [17] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI '06*, pages 308–319, New York, NY, USA, 2006. ACM.
- [18] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection., In *ICSE'09*, 2009 (to appear).
- [19] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.
- [20] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.
- [21] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 114–123, New York, NY, USA, 2005. ACM.
- [22] K. Sen. Race directed random testing of concurrent programs. In *PLDI'08*, volume 43, pages 11–21, New York, NY, USA, 2008. ACM.
- [23] J. Stasko. Animating algorithms with xtango. *SIGACT News*, 23(2):67–71, 1992.
- [24] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jrapture: A capture/replay tool for observation-based testing. *SIGSOFT Softw. Eng. Notes*, 25(5):158–167, 2000.
- [25] C. von Praun and T. R. Gross. Object race detection. *OOPSLA'01*, 36(11):70–82, 2001.